<알고리즘 요약 정리본>

3장 기본 정렬 (170320 to 170322)

알고리즘에서 제일 기초적인 개념이 바로 정렬이라고 강조하고 싶다. 그렇지만 정렬 알고리즘은 선택 정렬에서부터 쉘 정렬 등등 여러 가지의 정렬 알고리즘이 존재하는데 우리가 공부하는 부분은 우선 기본 정렬이다. 이에 대해서 자세히 공부해보는 기회가 되어보자.

1. 기초 정렬 알고리즘

기초 정렬 알고리즘에 대해서는 우리가 대학교 1학년 때 의미 없게 공부한 선택 정렬, 버블 정렬 등등을 들어봤을 것이다. 그렇지만 우리가 정렬 알고리즘을 어떻게 공부하든 간에 무슨 뜻인지에 대해 이해를 하고 공부를 하는 것을 목표로 잡는 것이 도움이 되겠다.

1-1) 기초 정렬 알고리즘의 정의

기초 정렬 알고리즘은 정렬을 접근하는데 있어서 기본적으로 변수들만 잘 이용하면 쉽게 돌릴 수 있는 정렬 알고리즘이라고 할 수 있겠다. 예를 들어 선택 정렬은 최솟값을 이용, 버블 정렬은 swap 메소드를 중점으로, 삽입 정렬은 기준 값을 이용해서 계산을 하는 원리로 생각할 수 있다.

기초 정렬 알고리즘의 장점은 변수, 메소드들만 제대로 이용하면 복잡하지 않고 간단한 알고리즘으로 시작해서 끝을 내는 알고리즘이라는 것이다. 단점으로서는 알고리즘의 복잡도가 워낙 장난이 아니라서 O(n^2)이란 복잡도가 나오게 된다...

1-2) 정렬에 대한 알고리즘 구축

우리가 정렬을 하는 경우에 값에 대해서만 생각을 할 수 있겠지만, 실질적으로 이는 int 형 배열에 대해서만 고려를 하기 때문에 정렬이 좆밥이라고 생각을 할 수 있겠지만, 실제로 데이터를 다루는데 있어서 여러 가지 정렬하는 방안이 존재한다. 예를 들어 우리가 동전을 정렬하는 경우(500원, 100원, 50원, 10원)에 동전에 있는 값은 금액의 수뿐만이 아니라 동전이 만들어진 년도에 대해서도 정렬을 할 수도 있겠다. 아니면 같은 금액에 대해서는 년도가 최신년도인 경우로 정렬하는 방법을 바꾸고, 년도가 같으면 금액의 크기에 따라서 정렬을 하는 방안이 있다. 이처럼 정렬을 하는 기준에는 오로지 하나의 값만 고려하는 것이 아니라 여러 데이터에 대해서 고려를 할 필요가 있다. 이는 차후에 배우는 객체 정렬 부분에서 다시 설명하겠다.

2. 선택 정렬(Selection Sort)

선택 정렬은 최솟값(내림 차순인 경우에는 최댓값) 개념을 이용해서 정렬을 하는 원리이다. 우리가 어렸을 때 카드 게임을 한 거와 같은 원리로 생각하면 쉽다.

2-1) 선택 정렬 사례

10, 8, 22, 4, 9, 14, 20, 3	8개의 int 형 배열

여기서 선택 정렬이 돌아가는 순서를 아래와 같이 작성해두겠다.

10 (8, 22, 4, 9, 14, 20, 3)	k=0인 경우
3 (8, 22, 4, 9, 14, 20, 10)	3과 10을 바꾼다.
3 8 (22, 4, 9, 14, 20, 10)	k=1인 경우
3 4 (22, 8, 9, 14, 20, 10)	4와 8을 바꿔준다.
3 4 22 (8, 9, 14, 20, 10)	k=2인 경우
3 4 8 (22, 9, 14, 20, 10)	8과 22를 바꾼다.
3 4 8 22 (9, 14, 20, 10)	k=3인 경우
3 4 8 9 (22, 14, 20, 10)	9와 22를 바꾼다.
3 4 8 9 22 (14, 20, 10)	k=4인 경우
3 4 8 9 10 (14, 20, 22)	10과 22를 바꾼다.
3 4 8 9 10 14 (20, 22)	k=5인 경우
3 4 8 9 10 14 (20, 22)	14 그대로이다.
3 4 8 9 10 14 20 (22)	k=6인 경우
3 4 8 9 10 14 20 (22)	정렬의 완성.

2-2) 선택 정렬 알고리즘

이처럼 복잡한 여정이었지만, 각 배열의 값마다 다음 인덱스부터 최솟값을 찾아가면서 자리를 바꿔가는 원리로 돌아가는 알고리즘이 선택 정렬이다. 이 알고리즘은 아래와 같이 작성이 가능하다.

```
public static void selectionSort(int[] arr){
    for(int k=0;k<arr.length-1;k++){ // 1
        int minIndex=findMin(arr, k);
        swap(arr, minIndex, k);
    }
}
```

```
public static int findMin(int[] arr, int start){
    int minIndex=start; // 2
    for(int k=minIndex;k<arr.length;k++){
        if(arr[minIndex]>arr[k]){
            minIndex=k;
        }
    }
    return minIndex;
}
```

```
public static void swap(int[] arr, int x, int y){
    int temp=arr[x];
    arr[x]=arr[y];
    arr[y]=temp;
} // 3
```

1 : 방금 우리가 선택 정렬이 대략 위처럼 돌아가는 것을 작성했을 때 분명 k는 0부터 배열 인덱스 마지막 이전까진 돌아간다는 뜻을 알 수 있다. 그래서 반복문에는 k=0, k<arr.length-1 라는 녀석이 들어간다는 사실을 알 수 있다. 여기서 반복문은 (배열의 크기 -1) 즉 n-1 만큼 findMindex까지 돌게 되어 시간 복잡도는 O(n^2)이 되겠다.

2 : 최솟값을 찾아내기 위해서 중요한 점은 최솟값을 나타내는 인덱스인 minIndex를 start로 잡고 시작을 한다. 그럼 처음 값에 대해서는 굳이 비교할 필요가 없기 때문에 바로 다음 값에 대해서 비교를 들어가게 되니깐 for 문에서 범위를 k=minIndex+1부터 마지막 인덱스까지 돌게 되어 최솟값 인덱스를 주어가는 원리로 생각하면 되겠다. 이 메소드의 시간 복잡도는 자동적으로 O(n)이 되겠다.

3 : swap 메소드는 최솟값 인덱스와 k와 바꿔주는 역할을 한다. 이의 시간 복잡도는 O(1)이 되지만 여기서 x, y의 순서는 엇깔려서 써도 크게 상관 없으니깐 이에 대해서 큰 걱정할 필요는 없다.

2-참조) 선택 정렬을 최댓값을 이용해서 정렬하기

물론 선택 정렬을 최솟값만으로 찾는 것은 아니다. 예를 들어 최댓값을 이용해서 새로이 정렬하는 방안이 있다. 이는 아래와 같이 작성하면 올바르게 돌아가겠다.

```
public static int findMax(int[] a, int end){
        int maxIndex=end;
        for(int k=maxIndex-1;k>=0;k--){
                if(a[k]>a[maxIndex]){
                        maxIndex=k;
                }
        }
        return maxIndex;
}
public static void swap(int[] arr, int a, int b){
        int temp=arr[a];
        arr[a]=arr[b];
        arr[b]=temp;
}
public static void selectionSort(int[] a){
        for(int k=a.length-1;k>0;k--){
        int maxIndex=findMax(a, k);
        swap(a, k, maxIndex);
        }
}
```

그렇지만 최댓값을 찾는 과정은 뒤에서부터 작성을 해줘야지 오름차순이 되니깐 반대로부터 뒤집어서 시작을 해야 된다는 점에 대해서 알아두고 넘어가면 되겠다.

3. 버블 정렬(Bubble Sort)

우리가 어렸을 때 버블보블이란 고전 게임을 해봤을 것이다. 공룡이 적을 향해 비눗방울을 불러서 터트리는 게임인 사실에 대해서는 알 수 있다. 그렇지만 공룡이 계속 비눗방울을 불게 되면 적들이 들어있는 방울과 같이 위로 올라가는 장면에 대해서 발견을 할 수 있다. 이를 터트리면 보너스 점수들이 많이 나오는 사실보다 중요한 것은 비눗방울처럼 위로 올라가는 원리(배열에선 오른쪽으로 서서히 바뀌는 원리)를 이용해서 소개한 정렬 알고리즘이 바로 버블 정렬이 되겠다.

3-1) 버블 정렬 사례

10, 8, 22, 4, 9, 14, 20, 3	8개의 int 형 배열

이 배열에 대해서 버블 정렬이 어떻게 돌아가는지에 대해 아래처럼 작성해보겠다.

(10, 8) 22, 4, 9, 14, 20, 3	k가 7인 경우
8 (10, 22) 4, 9, 14, 20, 3	그대로 둔다.
8, 10 (22, 4) 9, 14, 20, 3	둘을 바꾼다.
8, 10, 4 (22, 9) 14, 20, 3	둘을 바꾼다.
8, 10, 4, 9, (22, 14), 20, 3	둘을 바꾼다.
8, 10, 4, 9, 14, (22, 20), 3	둘을 바꾼다.
8, 10, 4, 9, 14, 20, (22, 3)	둘을 바꾼다.
8, 10, 4, 9, 14, 20, 3 (22)	22까지 정렬 완료

(8, 10) 4, 9, 14, 20, 3 (22)	k가 6인 경우
8 (10, 4) 9, 14, 20, 3, (22)	둘을 바꾼다.
8, 4 (10, 9) 14, 20, 3, (22)	둘을 바꾼다.
8, 4, 10 (9, 14) 20, 3, (22)	그대로 둔다.
8, 4, 10, 9 (14, 20) 3, (22)	그대로 둔다.
8, 4, 10, 9, 14 (20, 3) (22)	둘을 바꾼다.
8, 4, 10, 9, 14, 3 (20, 22)	20까지 정렬 완료

(8, 4) 10, 9, 14, 3 (20, 22)	k가 5인 경우
4, (8, 10), 9, 14, 3 (20, 22)	그대로 둔다.
4, 8 (10, 9) 14, 3 (20, 22)	둘을 바꾼다.
4, 8, 9 (10, 14) 3 (20, 22)	그대로 둔다.
4, 8, 9, 10 (14, 3) (20, 22)	둘을 바꾼다.
4, 8, 9, 10, 3 (14, 20, 22)	14까지 정렬 완료.

(4, 8) 9, 10, 3 (14, 20, 22)	k가 4인 경우
4 (8, 9) 10, 3 (14, 20, 22)	그대로 둔다.
4, 8 (9, 10) 3 (14, 20, 22)	그대로 둔다.
4, 8, 9 (10, 3) (14, 20, 22)	둘을 바꾼다.
4, 8, 9, 3 (10, 14, 20, 22)	10까지 정렬 완료.

(4, 8) 9, 3 (10, 14, 20, 22)	k가 3인 경우
4 (8, 9) 3 (10, 14, 20, 22)	그대로 둔다.
4, 8 (9, 3) (10, 14, 20, 22)	둘을 바꾼다.
4, 8, 3 (9, 10, 14, 20, 22)	9까지 정렬 완료

(4, 8) 3 (9, 10, 14, 20, 22)	k가 2인 경우
4 (8, 3) (9, 10, 14, 20, 22)	둘을 바꾼다.
4, 3 (8, 9, 10, 14, 20, 22)	8까지 정렬 완료

(4, 3) (8, 9, 10, 14, 20, 22)	k가 1인 경우
3 (4, 8, 9, 10, 14, 20, 22)	4까지 정렬 완료
(3, 4, 8, 9, 10, 14, 20, 22)	k가 0이면 모든 정렬 완료

이처럼 버블 정렬은 생각보다 좀 복잡한 면이 많을 것이다. 그렇지만 우리가 공부한 정렬 알고리즘에서 선택 정렬 다음으로 쉬운 정렬밖에 안 된다. 버블 정렬은 k의 순서대로 완료된 모습들을 모으면 아래와 같이 된다.

8,	10,	4,	9,	14,	20,	3	(22)
8,	4,	10,	9,	14,	3	(20,	22)
4,	8,	9,	10,	3	(14,	20,	22)
4,	8,	9,	3	(10,	14,	20,	22)
4,	8,	3	(9,	10,	14,	20,	22)
4,	3	(8,	9,	10,	14,	20,	22)
3	(4,	8,	9,	10,	14,	20,	22)
(3,	4,	8,	9,	10,	14,	20,	22)

버블정렬은 쉽게 이야기해서 역삼각형을 만들어가는 과정과 똑같다고 생각하면 쉽다. 이러한 알고리즘은 어떻게 구성이 되었는지에 대해 아래에서 알아보자.

3-2) 버블 정렬 알고리즘

```
public static void bubbleSort(int[] arr){
    for(int k=arr.length-1;k>0;k--){
        for(int l=0;l<k;l++){
            if(arr[l]>arr[l+1])
                swap(arr, l, l+1);
        }
    }
}
```

```
public static void swap(int[] arr, int x, int y){
    int temp=arr[x];
    arr[x]=arr[y];
    arr[y]=temp;
}
```

방금 전에 이용했던 선택 정렬보다 그렇게 복잡하지 않은 구성으로 살펴볼 수 있다. 버블 정렬에서 제일 중요한 것은 직각삼각형을 만들어가는 과정이라고 생각을 하면 쉽게 접근이 가능하다.

3-3) 버블 정렬에 대한 시간 낭비 고려
예를 들어 아래와 같이 이미 정렬된 배열을 살펴보자. 1, 2, 9, 20, 30. 그런데 이를 또 버블 정렬로 조건문을 반복하기에는 시간적으로 낭비라고 볼 수 있다. 그래서 우리는 boolean 대수를 이용해서 시간 낭비를 방지하도록 작성 할 수 있다. 이를 이용해서 아래와 같이 작성하면 이미 정렬된 배열에 대한 시간 복잡도는 O(n)이 나와 종료될 수 있고, 아닌 경우에는 O(n^2)로 노가다 뛸 수도 있다.

```
public static void bubbleSort(int[] arr){
    for(int k=arr.length-1;k>0;k--){
        boolean complete=true;
        for(int l=0;l<k;l++){
            if(arr[l]>arr[l+1])
                swap(arr, l, l+1);
                complete=false;
        }
        if(complete) break;
    }
}
```

4. 삽입 정렬(Insertion Sort)

예를 들어 우리가 좌석 버스를 타는 상황을 생각해보자. 요즘은 자리 지정제를 도입하는 바람에 입석은 통근 시간대에만 가능하여 통근 시간에 좌석버스를 탑승하는데 45명보다 더 많은 경우에 대해서 생각을 해보자. 그러면 이 버스를 타는데 있어서 복잡하지 않게 타려면 맨 앞에 선 사람이 뒷자리부터 차곡차곡 앉으면 탑승 수속이 원활해진다. 이러한 원리를 적용한 것이 바로 삽입 정렬이다.(필자가 개인적으로 좋아하는 정렬이기도 한다.) 삽입 정렬의 알고리즘은 어떻게 돌아가는지에 대하여 아래를 살펴보면서 공부를 해보자.

4-1) 삽입 정렬 사례

(10) **8**, 22, 4, 9, 14, 20, 3	k=1인 경우
(8 10) 22, 4, 9, 14, 20, 3	8이 뒤로 넘어감

(8 10) **22**, 4, 9, 14, 20, 3	k=2인 경우
(8 10 22) 4, 9, 14, 20, 3	22보다 큰 값이 없으니 그대로...

(8 10 22) **4**, 9, 14, 20, 3	k=3인 경우
(8 10 4 22) 9, 14, 20, 3	4가 뒤로 넘어감
(8 4 10 22) 9, 14, 20, 3	4가 뒤로 넘어감
(4 8 10 22) 9, 14, 20, 3	가장 작으니 맨 앞으로 넘어감...

(4 8 10 22) **9**, 14, 20, 3	k=4인 경우
(4 8 10 9 22) 14, 20, 3	9가 뒤로 넘어감
(4 8 9 10 22) 14, 20, 3	뒤로 넘어감
break;	이 뒤로는 정렬이 완료되어 다음으로 넘어감.

(4 8 9 10 22) **14**, 20, 3	k가 5인 경우
(4 8 9 10 14 22) 20, 3	뒤로 넘어감
break;	정렬이 완료되어 다음 값으로 넘어감

(4 8 9 10 14 22) **20**, 3	k가 6인 경우
(4 8 9 10 14 20 22) 3	뒤로 넘어감
break;	정렬이 완료되어 다음 값으로 넘어감

(4 8 9 10 14 20 22) 3	k가 7인 경우
(4 8 9 10 14 20 3 22)	뒤로 넘어감
(4 8 9 10 14 3 20 22)	뒤로 넘어감
…	…
(3 4 8 9 10 14 20 22)	총 7번의 과정으로 뒤로 넘어가게 된다.

이처럼 삽입 정렬의 구조로서는 정렬이 된 부분(소괄호에 있는 값들), 정렬이 안 된 부분(소괄호로 안 묶인 부분)으로 되어 있어서 집합 개념을 통해서 쉽게 정리가 된다. 이러한 알고리즘은 또한 break 문이 있기 때문에 선택 정렬, 버블 정렬보다 그나마 효율적인 문장으로 살펴볼 수 있다.

4-2) 삽입 정렬 알고리즘

```
public static void insertionSort(int[] arr){
    int k, l; // 1
    for(k=1;k<arr.length;k++){ // 2
        int compare=arr[k];
        for(l=k-1;l>=0;l--){ // 3
            if(arr[l+1]>compare){ // 3
                arr[l+1]=arr[l]; // 3-1
            }
            else break;
        }
        arr[l+1]=compare;
    }
}
```

1 : 우선 for 문 내부에서 l를 그대로 쓸 필요성이 존재하기 때문에 밖에다가 선언을 해서 초기화를 해야할 필요가 있다. 물론 k 반복문 내부에 작성해도 되는데 이러한 방안이 알고리즘의 효율성을 위해 좋은 방안으로 생각을 하면 좋겠다.

2 : k가 0인 경우에는 굳이 비교할 필요가 없기 때문에 1부터 마지막 값까지 비교를 해서 뒤에다가 넣어가는 개념으로 생각하면 되겠다. 여기서 compare는 인덱스가 k인 값에 대해서 아까 설명한 정렬이 된 부분 집합에 넣기 위한 값을 찾는 과정을 위해 필요로 한 비교 값으로 생각하면 되겠다.

3 : 이 과정은 compare보다 큰 값들은 앞으로 보내는 과정이다. 또한 어차피 뒷부분만 비교하면 되기 때문에 l-1에서 시작해서 반복문을 돌면 된다. 가령 예를 들어 4-1에서 k가 3인 경우에 잠깐만 살펴보도록 하자. 다음 쪽에서 상세히 설명하겠다.

k=3인 경우에 compare는 4가 된다.

(8 10 22) **4**, 9, 14, 20, 3	compare=4
(8 10 22 22) 9, 14, 20, 3	l은 2이 되고 arr[3]은 22로 임시.
(8 10 10 22) 9, 14, 20, 3	l은 1이 되고 arr[2]는 10으로 임시.
(8 8 10 22) 9, 14, 20, 3	l은 0이 되고 arr[1]은 8로 임시.
l은 결국 -1이 되어서 반복문을 벗어나게 된다.	
(4 8 10 22) 9, 14, 20, 3	arr[-1+1]=compare; 즉 arr[0]이 4로 되어 종결.

물론 삽입 정렬을 하더라도 정렬된 배열에 대해서 시간 복잡도가 낭비되는 현상도 존재하긴 하지만 선택 정렬, 버블 정렬과 마찬가지로 시간 복잡도는 O(n^2)이 되겠다.

---

5. 자바에 있는 Arrays 클래스

자바에도 이러한 정렬 과정을 위한 메소드가 숨겨져 있다는 사실을 알 것이다. 이 메소드에 대해서 자바를 공부하는데 있어서 알아두면 약이 되거나 총알이 될 수도 있다.

기본적으로 Arrays 클래스에 존재하는 메소드들, 맴버 변수에 대해 알아두면 약이 되는 개념들을 짚고 넘어가자.

- public static String toString(부지기수한 배열);

toString은 아시다시피 객체 내부의 맴버 변수들을 출력하는데 있어서 흔히 쓰이는 메소드이다. 예를 들어 배열 arr의 요소들을 출력할 때는 arr.toString()을 쓰는 것이 아니라 Arrays.toString(arr)를 써야 된다. 이는 static 메소드이기 때문이다.

- public static void sort(부지기수한 배열);

이는 말 그대로 배열 안에 있는 값들을 오름차순으로 정렬시켜주는 역할을 한다. 물론 디폴트 값 기준으로는 오름차순을 해주지, Comparator를 이용하면 내림차순 정렬도 가능하다는 점을 알아두자.

- public static void sort(부지기수한 배열, 시작 인덱스, 종료 인덱스)

이처럼 작성을 하면 만일 Arrays.sort(arr, 0, 3)을 쓰면 0번째 인덱스부터 2번째 인덱스까지 정렬을 시켜주는 역할을 한다. 자바에서는 대부분 시발점과 종착점이 특이한데 거의 시발점<=인덱스<종착점으로 되어 있다는 점을 알아두고 가면 좋겠다.

- public int length;

이는 배열의 길이이다. 맴버 변수이긴 하지만, public형 변수라서 arr.length만으로도 쉽게 배열의 길이를 가져올 수 있다.

- public static int binarySearch(부지기수한 배열, 찾을 값)

이는 우리가 쉽시간에 다룬 이진탐색을 구현한 메소드이다. 물론 부지기수한 배열의 데이터 형식과 찾을 값의 데이터 형식은 같아야 되는건 아시죠?

- public static void fill(부지기수한 배열, 채울 값)

배열의 크기를 초기화하였을 경우에만 배열에 채울 값을 모두 채울 수 있다. 예를 들어 크기가 10인 배열에 10을 채우고 싶음 Arrays.fill(arr, 10)을 작성하면 되겠다. 물론 범위를 잡아서 0번째 인덱스부터 5번째 인덱스까지 모두 20으로 채우고 싶다면 Arrays.fill(arr, 0, 6, 20)으로 쓰면 되겠다. 범위는 아까와 같다고 생각하면 된다.

<추가로 알아야 하는 상식> 우리가 흔히 쓰는 int[], double[] 등등의 배열에는 우리가 서랍에 10000원짜리 지폐를 넣으면 시중에서 물건 사는데 유용하게 쓸 수 있지만, Integer[], Double[] 등등 래퍼 클래스나 사용자임의 클래스로 호출된 객체는 서랍에 1억짜리 수표가 들어있어서 돈을 은행에서 바꿔서 써야 되듯이 객체의 참조 주소가 들어있다는 사실에 대해 알고 넘어가야겠다. 하기야 서랍 안에는 현찰로 1억원을 모두 못 채워서 넣듯이 자바에서는 객체를 실질적으로 저장하진 못한다. 또한 자바에서는 배열도 객체 취급을 한다. 이로써 배열은 heap 저장 공간 영역에 생성이 되는 사실을 알고 넘어가자.

6. Comparable 인터페이스

Comparable 인터페이스에는 compareTo() 라는 메소드가 있다. 이를 이용해서 this 객체와 매개변수 객체끼리 비교해 크면 양수 출력, 적으면 음수 출력, 같으면 0을 출력하게 된다.

```
inferface Comparable<T>{
    int compareTo(T obj){ ... }
}
```

여기서 <T>는 아시다시피 제네릭 클래스의 개념이다. 모르겠다면 무슨글이나 뭐시기버에다가 검색해보면 되겠다.

우리가 예를 들어 Coin이란 객체에 Comparator<Coin> 인터페이스를 이용해보도록 하자.

```
class Coin implements Comparable<Coin>{
    private int won;
    private int year;
    ...(getter/setter 생략)
    @Override
    public int compareTo(Coin coin){
  int wonComp=this.getWon()-coin.getWon();
  if(wonComp!=0) return wonComp; // 1
else return this.getYear()-coin.getYear(); // 2
    }
}
```

1 : wonComp는 this 객체에 있는 돈 단위와 coin 객체에 있는 돈 단위를 비교하는 과정이다. 만일 같은 금액의 동전이면 2번 문장을 진행하게 되고, 다른 금액이면 금액의 차이를 반환하게 되어 음수이면 앞 부분에 정렬, 크면 뒷 부분에 정렬하게 된다.

2 : 동전의 금액이 같은 경우에는 년도를 비교해서 오래된 동전이면 앞부분에 최근에 나온 동전이면 뒷부분에 정렬이 되는 방식이다. 이러한 문장을 통해서 동전의 금액에 따라, 같으면 연도에 따라 비교를 해주는 클래스로 살펴보면 되겠다.

7. Comparator 인터페이스

6번에서 이야기한 Comparable은 애석하게도 순서를 지켜서 할 수 밖에 없었다. 가령 우리가 동전을 비교하는 경우에 금액을 기준으로 하는 것이 아니라 발행

연도에 따라서 비교를 하는 경우도 있고, 금액의 내림차순 대로도 비교를 할 수 있다. 사람도 마찬가지이다. 이름, 나이, 키 등등... 하지만 순서 기준이 래퍼 클래스나 Date 클래스 등등 이외에 여러 개로 잡히는 경우에는 Comparator를 따로 구현해서 정렬을 하는 방안이 최고이다. Comparator 인터페이스는 아래와 같이 구성되어 있다.

```
inferface Comparator<T>{
    int compare(T obj1, T obj2){ ... }
}
```

물론 마찬가지로 어느 맴버 변수의 기준을 잡아서 obj1이 obj2보다 큰 경우에는 양수를 출력, 같으면 0, 다르면 음수를 출력하는 원리로 생각하면 되겠다.

이번에는 동전의 연도나 금액에 따라서 비교를 다르게 비교를 하는 메소드를 추가해보자.

```
import java.util.Comparator; // 1
class Coin{
    private int won;
    private int year;
    ...(getter/setter/toString 생략)
    }
}
public class CoinWonComparator implements
Comparator<Coin>{
  @Override
  public int compare(Coin c1, Coin c2){
    int wonComp=c1.getWon()-c2.getWon();
    if(wonComp!=0) return wonComp;
    else return c1.getYear()-c2.getYear();
  } // 2
}
public class CoinYearComparator implements
Comparator<Coin>{
  @Override
  public int compare(Coin c1, Coin c2){
    int yearComp=c1.yearWon()-c2.yearWon();
    if(yearComp!=0) return yearComp;
    else return c1.getWon()-c2.getWon();
  } // 3
}
```

1 : Comparator는 util 패키지 내부에 존재하는 클래스이기 때문에 처음에 import를 해줄 필요가 존재한다. 이처럼 작성을 하고 나면 나머지는 Comparable이랑 같지만 이에 대한 차이점은 기술적으로 숙지하고 넘어가면 되겠다.
2 : wonComp는 금액에 따라서 비교를 하는 것인데 금액별로 비교를 하고 금액이 같으면 연도별로 비교를 하는 compare 오버라이딩 메소드이다.
3 : yearComp는 연도에 따라서 비교를 하는 것인데 연도별로 비교를 하고 연도가 같으면 금액별로 비교를 하는 compare 오버라이딩 메소드이다.

이처럼 comparator 인터페이스를 이용해서 다형성을 구축한 원리로 오버라이딩을 하면 객체 별로 비교해야 하는 맴버 변수에 대해 순서가 확립되기 때문에 알아두면 좋겠다. 배열에서 정렬은 아래와 같이 진행된다.

```
import java.util.Arrays;
public static void main(String[] args){
    Coin[] wallet={new Coin(100, 1990), new
Coin(100, 1993), new Coin(50, 1998), new
Coin(500, 1997), new Coin(10, 1990)};
Arrays.sort(wallet, new CoinYearComparator());
System.out.println(Arrays.toString(wallet));//1
Arrays.sort(wallet, new CoinWonComparator());
System.out.println(Arrays.toString(wallet));//2
}
```

여기서 1번에 나오는 문장이랑 2번에 나오는 문장이랑은 천지만별 차이이다. 우선 1번 문장에 나오는 결과는 아래와 같다.
[10원 1990년], [100원 1990년], [100원 1993년]
, [500원 1997년] [50원 1998년]
이는 연도가 같은 경우에는 낮은 금액 먼저 출력이 되고, 연도 오름차순으로 출력이 되겠다.
하지만 2번 문장에서는 다음과 같이 나온다.
[10원 1990년], [50원 1998년], [100원 1990년]
, [100원 1993년], [500원 1997년]
이는 금액이 같은 경우에는 연도가 오래된 것부터 먼저 출력이 되고, 금액 오름차순으로 출력을 한 것이다.
물론 내림차순을 하고 싶다면
c1.getWon()-c2.getWon()을
c2.getWon()-c1.getWon()으로 살짝만 바꿔줘도 내림차순이 가능하다는 점을 알고 넘어가자.

<참조1> 래퍼 클래스가 필요로 한 절대적인 이유는 무엇일까?
래퍼 클래스는 dok2나 지코처럼 랩하는 래퍼 클래스가 아니라 예를 들어 int형 변수, double형 변수에 대해서 자료구조형을 구현한 Collection 클래스에서 주로 쓰이는 것이 객체인데 이를 구현하기 위해서는 어쩔 수 없이 객체를 만들 수 있는 클래스가 필요하다. Collection 클래스를 위해서 만들었다고 생각하면 되겠다.

<참조2> 상속과 인터페이스의 특징을 짚고 넘어가자.
공통점 : 다형성을 구현해서 각 객체에 존재하는 메소드들에 대해서 어느 객체를 실행하든 간에 호환성을 보장하는 역할을 한다.
차이점 : 상속은 이를 받은 객체에만 메소드를 구현할 수 있고, 자바의 체계에서는 부모 객체의 수가 1개일 수밖에 없는 상속을 받게 된다.(물론 C++를 공부해보면 friends나 상속의 막장판을 볼 수 있지만...) 하지만 인터페이스는 상속이란 개념보다는 우리가 부재중인 경우에 핸드폰으로 카톡이나 연락을 취할 수 있는 통신 기기같은 역할로 보면 되겠다.